$\mathcal{WIN}$: An Efficient Data Placement Strategy for Parallel XML Databases

Nan Tang† Guoren Wang‡ Jeffrey Xu Yu† Kam-Fai Wong† Ge Yu‡

† Department of SE & EM, The Chinese University of Hong Kong, Hong Kong, China
‡ College of Information Science and Engineering, Northeastern University, Shenyang, China
{ntang, yu, kfwong}se.cuhk.edu.hk {wanggr, yuge}@mail.neu.edu.cn

Abstract

The basic idea behind parallel database systems is to perform operations in parallel to reduce the response time and improve the system throughput. Data placement is a key factor on the overall performance of parallel systems. XML is semistructured data, traditional data placement strategies cannot serve it well. In this paper, we present the concept of intermediary node INode, and propose a novel workload-aware data placement $\mathcal{WIN}$ to effectively decluster XML data, to obtain high intra query parallelism. The extensive experiments show that the speedup and scaleup performance of $\mathcal{WIN}$ outperforms the previous strategies.

1 Introduction

XML has become the *de facto* standard for data representation and exchange over the Internet. Nevertheless, along with the increasing size of XML document and complexity of evaluating XML queries, existing query processing performance in concentrative environment will get a limitation. Parallelism is a viable solution to this problem.

In parallel DBMSs, data placement has a significant impact on the overall system performance. Several earlier studies have shown that data placement is closely relevant to the performance of a shared-nothing parallel architecture. Many classical data placement strategies are thereby proposed in the literature. [2,11] proposed many single dimension and multiple dimension data placement strategies for structured data in RDBMSs. In OODBMSs otherwise, since the objects can be arbitrarily referenced by one another, the relationship of objects is regarded as a graph on topology. Thus most of the object-placement strategies are graph-based.

Distinct from them, XML employs a tree structure. Conventional data placement strategies in RDBMSs and OODBMSs cannot serve XML data well. Semistructured data involve nested structures; there are much intricacy and redundancy when representing XML as relations. Data placement strategies in RDBMSs cannot, therefore, well solve the data placement problem of XML data. Although graph partition algorithms in OODBMSs can be applied to tree partition problems, the key relationships between tree nodes such as ancestor-descendant or sibling cannot be clearly denoted on graph. Hence data placement strategies for OODBMSs cannot well adapt to XML data either. Furthermore, the path based XML query languages such as XPath and XQuery enhance the hardness and intricacy of tree data placement problem. Data placement strategies in both parallel relational and object-oriented database systems are helpful to the data placement for XML documents. They cannot, nevertheless, solve the problem which XML faces. Efficient data placement strategies for XML DBMSs are, therefore, meaningful and challenging.

[13] proposed two data placement strategies for XML data tree: NSNRR and PSPIB. NSNRR is proposed to exploit pipeline query parallelism. With NSNRR elements having different tag names are allocated to different sites in a round-robin way. Previous experiments show that when the document is small NSNRR has good speedup and scaleup performance; but as the document grows, both the speedup and scaleup performance of NSNRR behave badly. PSPIB is a data placement strategy based on path instances. Path instance is informally defined as a sequence of node identifiers from root to leaf node, while path schema is an abstract of path instance. The main idea of PSPIB is to decluster the path instances having

the same path schema evenly across all sites. Because PSPIB and $\mathcal{WIN}$ have the similar idea to partition the XML data tree to form sub-trees, we compare $\mathcal{WIN}$ with PSPIB in the performance evaluation. We give a simple example of data distribution of PSPIB to make it more concrete. Given a lightweight XML data tree in Fig. 1, Fig. 2 shows the data placement under the case of two sites. Solid circles represent the elements duplicated at both sites; real circles represent the elements distributed to site 1; dotted circles represent the elements distributed to site 2.

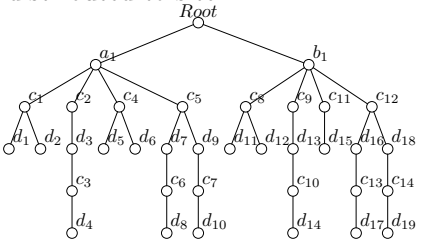


Figure 1. A Lightweight XML Data Tree

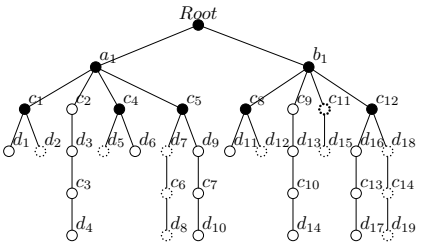


Figure 2. Data Placement of PSPIB

In this paper, we present the concept of intermediary node $INode$. Considering that determining the best data placement strategy in parallel DBMSs is an NP-complete [9] problem, a novel workload-aware data placement strategy $\mathcal{WIN}$ is proposed to find the sub-optimal $INode$ set for partitioning XML data tree. This strategy is not aim at special queries, nevertheless, as will be shown, this strategy suits each query expression for good intra query parallelism. Based on an XML database benchmark–XMark [10], comprehensive experimental results show that our new data placement strategy $\mathcal{WIN}$ has better speedup and scaleup performance than previous work.

The rest of this paper is organized as follows. Section 2 gives the concept of $INode$ and the basic idea for data placement. Section 3 provides a detail statement of the novel data placement strategy $\mathcal{WIN}$. Section 4 shows experimental results. Our conclusions and future work are contained in Section 5.

2 Data Placement using INode

Observe from a common XML data tree, we find that the number of upper level nodes is always few. And yet these nodes are the crux of many XML query expressions. Therefore, we should duplicate these nodes to all sites; the tiny storage space consumption can trade off between high intra query parallelism and low communication overhead. We will partition the remaining sub-trees to different sites, and then in each site, combine the sub-trees and duplicated nodes to form a local data tree. $\mathcal{WIN}$ describes how to find a group of duplicate nodes and a group of sub-trees to tradeoff the duplication cost and high parallelism. The problem is formally defined as follows:

Let XT be an XML data tree, Q be the query set of n XML query expressions $Q_1, Q_2, \cdots, Q_n$, and f_i be the frequency of Q_i, let $N > 1$ be the number of sites. How to partition XT to N sub-trees $XT_1 \sim XT_N$, $XT = \bigcup_{i=1}^{N} XT_i$, to take each Q_i a high intra query parallelism.

We will first present the concept of $INode$, and then give the basic data placement idea based on $INode$, last we give our workload estimation formulae.

2.1 Intermediary Node

We classify nodes in an XML data tree XT into three classes: Duplicate Node ($DNode$), Intermediary Node ($INode$) and Trivial Node ($TNode$). $DNode$ will be duplicated at all sites. $INode$ is the essential node that decides which nodes will be partitioned to different sites. $TNode$ will be partitioned to the site where their ancestor $INode$ is. We use DN to represent the set of $DNodes$, IN for $INodes$ and TN for $TNodes$. They satisfy the following properties:

1. $DN's$ ancestor nodes are DN or $NULL$;
2. $IN's$ ancestor nodes are DN or $NULL$;
3. $IN's$ descendant nodes are TN or attribute nodes or text nodes or NULL;
4. DN is duplicated in all sites, IN, TN are unique;
5. $IN \cap DN \cap TN = \emptyset$;
6. $IN \cup DN \cup TN = XT$.

DN forms the duplicated root tree in all sites, and each $INode$ in IN is the root of a sub-tree to be partitioned. IN is the pivot of our data placement strategy. For a certain IN, our partition algorithm will get one data placement result. Different IN will conduct different data placement, the key of our novel data placement strategy is how to find the best IN. There are numerous IN in a data tree, as shown in Fig. 3, $\{b, c\}$ is a possible $INode$ set, and also $\{b, f, g\}$ and $\{d, e, c\}$, $INode$ could be leaf node, e.g., $\{b, l, m, g\}$, $\{b, f, n, o\}$, $\{b, l, m, n, o\}$, etc. The extreme case of IN is all $INodes$ are leaf nodes such as $\{h, i, j, k, l, m, n, o\}$. How to choose the best IN is obvious a NP-Complete

problem, as discussed in [9] which search the best data placement strategy in parallel DBMSs.

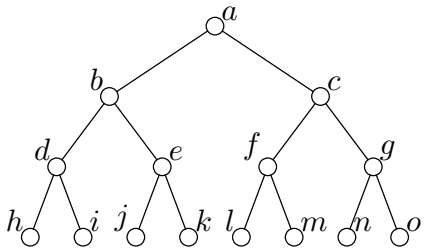


Figure 3. Example for INode

2.2 Data Placement based on INode

The basic idea of data placement based on $INode$ is described below. Given an XML data tree XT and an $INode$ set IN, the ancestors of all INodes are DN, and the remaining nodes in XT are TN. We first duplicate the root tree formed by DN in all sites, then we group the $INode$s in IN according to their parent node, in each group, we distribute the sub-trees rooted with $INode$s to different sites with the goal of workload average. On each sites, combine all the sub-trees and DN to form one local data tree.

Our goal is high intra query parallelism, thereby low system response time and high throughput. However, if there are many complex query expressions, there is no any data placement which can make each query having equal workload in all sites. Moreover, we cannot analyze all possible data placement results to find the best, which is obviously impractical. We exploit two heuristic rules to reduce the search space which are discussed in Section 3.

2.3 Workload Cost Model

In workload-aware data placement strategies, workload estimation is a crucial part since the outcome of workload estimation will directly affect the data placement result, thereby influence the performance of a parallel system. In this section we propose a cost model to estimate the workload of XML data tree in a shared-nothing parallel database system. These formulae are based not only on the inherent structure of XML data and query expressions but also on the index structures and query methods. Different from the path indices for regular path expressions proposed in [4], we only employ basic indices implemented in XBase [7]. As for some core problems such as ancestor-descendant query and twig join, we implement them with query rewriting and hash join technology though they have been efficiently solved in [1, 5]. The reason is that we aim at proposing a workload estimation model to meet demand for common XML query processing techniques instead of certain algorithms or specific systems.

Now we will give our workload cost model to evaluate the workload of a data tree with root a. Table 1 gives the variables used in this paper.

Variable	Description
n_a	the number of element a
N	the number of sites
$n_{a \oslash b}$	the result number of a $\oslash$ b
S_a	the size of element a
Q_i	the i-th query in statistics
f_i	query frequency of Q_i
$\delta_{a \oslash b}$	selectivity of operation a $\oslash$ b
$t_{I/O}$	avg. disk access time
v_{net}	net speed
W_a	workload of tree rooted a
$W_{label(a,b)}$	workload of label(a,b)
$f_{label(a,b)}$	frequency of label(a,b)

Table 1. Variables used in this paper

We define the workload of a data tree with root a as:

$$W_a = \begin{cases} 0 & \text{a is leaf} \\ \sum\limits_{b \in child(a)} (W_b + W_{label(a,b)}) & \text{a is not leaf} \end{cases} \tag{1}$$

where $W_{label(a,b)}$ is the workload of label from a to b.

In an XML data tree, the label workload from element a to b is:

$$\begin{aligned} W_{label(a,b)} &= \left(\frac{PreSize + TempSize}{S_{page}} \cdot t_{I/O} \right. \\ &\left. = + \frac{TempSize}{v_{net}} \right) \cdot f_{label(a,b)} \end{aligned} \tag{2}$$

The first item in the bracket is the cost of disk I/O, including the preprocessing course of hash table and storage of temporary results. The second is the communication cost for transferring temporary results. Consider the *cpu* cost is in a higher order of magnitude than disk I/O and net communication, we ignore it here. And $f_{label(a,b)}$ is the number of $label_{(a,b)}$.

The PreSize of constructing and using hash table is:

$$PreSize = (3n_a + 3n_b) \cdot S \tag{3}$$

it shows the size of constructing and using hash table. a and b has the same size in our OODBMS, so we use S to represent the object size of a and b when clear.

The temporary results size TempSize is defined as:

$$TempSize = n_b \cdot \delta_{a \oslash b} \cdot (S_a + S_b) \tag{4}$$

it shows the size of temporary results of $a \oslash b$. Here operation "$\oslash$" mainly refers to parent-child join "/".

The total number of $label(a, b)$ is defined as:

$$f_{label(a,b)} = \sum_{i=1}^{n} \Big(\big(Q_i \lhd label(a,b)\big) \cdot f_i \Big) \qquad (5)$$

where $Q_i \lhd label(a,b)$ represents the existing number of $label_{(a,b)}$ in query Q_i.

Selectivity $\delta_{a \oslash b}$ is defined as:

$$\delta_{a \oslash b} = n_{a \oslash b} / n_b \qquad (6)$$

Selectivity represents the percent of $n_{a \oslash b}$ with b after the operation $\oslash$.

3 $\mathcal{WIN}$ Data Placement Strategy

In this section we present the framework of our workload-aware intermediary node ($\mathcal{WIN}$) data placement strategy. We first give the holistic algorithm of $\mathcal{WIN}$, and then we will discuss each part in detail one by one. The following shows the comprehensive description of how to find the sub-optimal IN and how to get the data distribution. We consider only element nodes in the $\mathcal{WIN}$ algorithm because attribute nodes and text nodes are partitioned according to their parent element nodes.

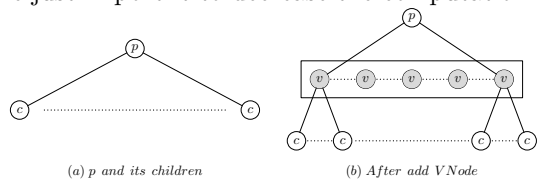
The first algorithm outlines $\mathcal{WIN}$ algorithm for fragmenting an XML data tree into partitions. In this algorithm, a parameter XT represents an XML data tree to be partitioned, a parameter N represents the number of sites, *interList* represents the list set of $INode$ and $finalIN$ the result of IN. Briefly, the algorithm creates each data placement as follows:

```
WIN(XT, N)
1  interList ← {{root}}
2  finalIN ← NIL
3  while interList ≠ NIL
4     do IN ← interList.popFront()
5        if BENEFIT(IN, finalIN)
6           then finalIN ← IN
7        EXPAND(interList, IN)
8  return PARTITION(finalIN)
```

Next procedure outlines PARTITION () for distributing an data tree with a given $INode$ set IN to different sites. With the view to get high intra query parallelism, combining with the special structure of XML document and the characters of query expressions, we first group nodes in IN with the same parent with the goal that the sub-trees in each group have the similar structure, and then partition sub-trees in each group to different sites with workload balancing to guarantee high intra query parallelism. After this procedure, each site has a local XML data tree.

```
PARTITION(IN)
1  DN ← ⋃_{E∈IN} Ancestor(E)
2   duplicate each node in DN to all sites P1toPN
3  for each element E ∈ IN
4     do group E according to getParent(E)
5  for each group G
6     do num ← 1
7        for each element E ∈ G
8           do workload_AVG ← W_(getParent(E))/N
9              workload_ADD ← 0
10             if workload_ADD < workload_AVG
11                then workload_ADD += W_E
12                else workload_ADD ← 0
13                     num ← num + 1
14             distribute Subtree(E) to P_num
15 return distribution of IN
```

The following procedure describes the EXPAND () procedure which expands *interList*. Each time we only expand an $INode$ set IN in *interList*, the total expand number is $|IN|$ ($|IN|$ represents the node number of IN). Each time we pop a new node $in \in IN$ and add all its children to IN for expanding, this procedure only expands those results which could take more benefit in IN and add them to *interList*. Suppose, for example, that a node has numerous children, e.g., 10000, the expansion of this node will definitely take a time-consuming operation, and further expansion is unthinkable. As shown in Fig. 4(a), the parent node p has many children nodes c, if we expand p, then the *interList* will grow huge. Under this case, we add the virtual node called $VNode$ to solve this problem. $VNode$ is the node between p and its children c. The gray nodes in Fig. 4(b) are $VNodes$. With $VNode$ we could solve above problem. We do not store $VNode$ while just import it to decrease the computation.



(a) p and its children *(b) After add VNode*

Figure 4. An Example of VNode

```
EXPAND(interList, IN)
1  for each node in ∈ IN
2     do IN' ← IN − in ⋃ in.getChildren()
3        if BENEFIT(IN', IN)
4           then interList.pushBack(IN')
5  return interList
```

Search all possible chances to find the best result is an NP-complete problem, so we add two heuristic rules in computing whether new $INode$ set IN_1 can take more benefit than the old $INode$ set IN_2 and judge to push it or not, with which to reduce the search space. It is shown in the procedure BENEFIT (). From

EXPAND () procedure, we can see that after each expansion of IN, there is more DN of new IN. There is a trend that with the further expansion, the workload distribution is more average, but the workload in each site is higher with the increasing of duplication cost. The extreme case is full duplication. Numerous studies have shown that full duplication leads to reduced performance and this has lead many to believe that partial duplication is the correct solution. Therefore we exploit two heuristic rules to simplify search space and avoid full duplication. They are shown below:

1. If IN_1 takes worse workload distribution than IN_2, similar to hill climbing method, we assume that there is no meaning to continue expanding and we do not expand in this case;
2. Even if IN_1 takes better workload distribution than IN_2, the duplication cost is higher than the benefit it takes; we also do not expand in this case.

BENEFIT(IN_1, IN_2)
1 PARTITION(IN_1)
2 $w_1, w_2, \ldots, w_N$ is $IN_1's$ workload from P_1 to P_N
3 PARTITION(IN_2)
4 $w'_1, w'_2, \ldots, w'_N$ is $IN_2's$ workload from P_1 to P_N
5 $W_{IN_1} \leftarrow \sum_{i=1}^{N} w_i$
6 $W_{IN_2} \leftarrow \sum_{i=1}^{N} w'_i$
7 $Avg_1 \leftarrow W_{IN_1}/N,\ Avg_2 \leftarrow W_{IN_2}/N$
8 $Max_1 \leftarrow max(w_1, w_2, \ldots, w_N)$
9 $Max_2 \leftarrow max(w'_1, w'_2, \ldots, w'_N)$
10 $Ben_1 \leftarrow Max_1 - Avg_1,\ Ben_2 \leftarrow Max_2 - Avg_2$
11 **if** $Ben_1 \geq Ben_2$
12 **then return** $false$
13 **else** $\Delta_{ben} \leftarrow Ben_2 - Ben_1$
14 $\Delta_{dup} \leftarrow (W_{IN_1} - W_{IN_2})/N$
15 **return** $(\Delta_{ben} - \Delta_{dup}) > 0$

4 Performance Evaluation

$\mathcal{WIN}$ and PSPIB are both based on the idea to partition path instances to form sub-trees, $\mathcal{WIN}$ is aim at workload balance and the goal of PSPIB is the average of path instances. To compare these two data placement strategies, therefore, we implemented them in a native XML database and completed performance test. In this section, we present an experimental evaluation of our novel workload-aware data placement strategy $\mathcal{WIN}$ and PSPIB. We start with a description of our experiment platform and test data set in Section 4.1. Section 4.2 presents a result evaluation and performance evaluation of our algorithms.

4.1 Experiment Platform and Test Data

The experiment platform used is a Network of Personal Computers–NOPC. There are six nodes; each node has a PIII 800HZ CPU, 128MB of memory and a 20G hard disk. The operating system used is Sun Unix(Solaris8.0) Intel platform edition of Sun Microsystems. They form a shared-nothing parallel XML database system that communicates through 100Mb/s high-speed LAN. One of them acts as scheduler and the others as servers. We employ a distributed and parallel OODBMS FISH [12] to store XML data. The testing programs were written in Forte C++ and INADA conformed to ODMG C++ binding. Because of establishment limitation, we can only use six nodes to test now, however, our algorithm could be easily extended to more nodes and our current experiment could basically show the performance trend. And we will give the test of more nodes in the future.

We adopt XML Benchmark Project [10] proposed by German CWI as our experiment data set. The XMark data set provides a large single scalable document; each is modelled after the same DTD given in [10]. The test data consists of 20M, 40M, 60M, 80M and 100M scaling single XML document. We use a comprehensive query set of 20 queries [10] to test the performance of these two data placement strategies.

4.2 Experimental Results and Performance Evaluation

$\mathcal{WIN}$ *vs* PSPIB: In this section, we evaluate the performance of $\mathcal{WIN}$ strategy and compare it with PSPIB. To assure the impartial performance evaluation of these two data placement, we use the same query method for these two strategies. With the index structures PNameIndex, NameIndex, TextValueIndex and AttributeValueIndex proposed in [8], we adopt query rewriting and hash join operation for all query expressions. To make experimental results more accurate, we executed each query expression 5 times, the first time is cold results and the latter is hot results. The average time of the hot results is the final response time. Since there are 5 test documents from 20M to 100M, each is tested from 1 site to 5 sites for all queries, the test work and experimental results are very large. We evaluate two key properties in measuring a parallel database management system: speedup and scaleup. In view of the memory limitation, when handling huge XML documents such as 80M and 100M, one site is always the exception point in performance evaluation. In speedup performance evaluation, therefore, we begin with the case of two sites. Not all performance curves are given

here because of the space limitation.

We first give the performance curves of Q_5, this query is to test the casting ability. Strings are the generic data type in XML documents [10]. Queries that interpret strings will often need to cast strings to another data type that carries more semantics. The query requirement is shown below:

How many sold items cost more than 40?

This query challenges the DBMS in terms of the casting primitives it provides. Especially, if there is no additional schema information or just a DTD at hand, casts are likely to occur frequently.

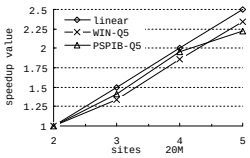


Figure 5. Q_5 speedup of 20M document

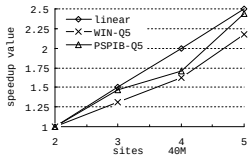


Figure 6. Q_5 speedup of 40M document

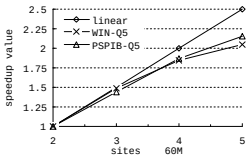


Figure 7. Q_5 speedup of 60M document

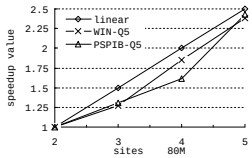


Figure 8. Q_5 speedup of 80M document

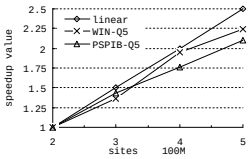


Figure 9. Q_5 speedup of 100M document

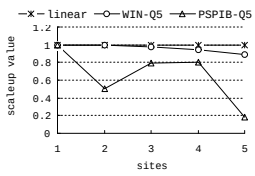


Figure 10. Q_5 scaleup

The speedup curves of Q_5 under two partition strategies were shown in Fig.5–Fig.9. It is easy to see that in this case both strategies have good speedup performance. It demonstrates that our native XML database has good support for casting primitives. Q_5 contains only single path expression, there is few cost for results combination and in most case, the local query results in both strategies are final results. Therefore, both strategies have good speedup performance.

Both $\mathcal{WIN}$ and PSPIB's scaleup curves of Q_5 were shown in Fig.10. One site testes 20M document, two sites 40M document, three sites 60M document, four sites 80M document and five sites 100M document. From it, we can clearly see that: with the increasing of the site number, the scaleup curve of $\mathcal{WIN}$ decreases smoothly and is close to linear. On the contrary, with the going up of the site number, the scaleup curve of PSPIB changes irregularly and embodies a vibrate change trend. So we can get the conclusion: in this query model, $\mathcal{WIN}$ has better scalability than PSPIB.

Next we give the performance analysis of Q_{17}. This is to test how well the query processors knows to deal with the semi-structured aspect of XML data, especially elements that are declared optional in the DTD. The query requirement is shown below:

Which persons don't have a homepage?

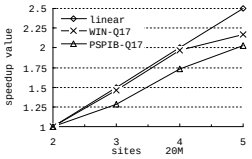


Figure 11. Q_{17} speedup of 20M document

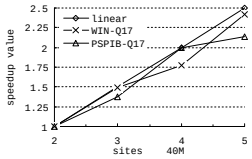


Figure 12. Q_{17} speedup of 40M document

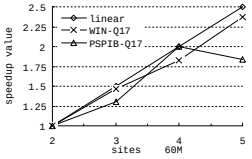


Figure 13. Q_{17} speedup of 60M document

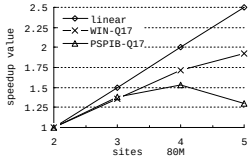


Figure 14. Q_{17} speedup of 80M document

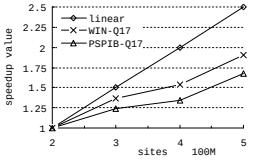


Figure 15. Q_{17} speedup of 100M document

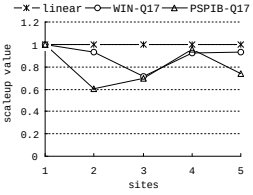


Figure 16. Q_{17} scaleup

Homepage is a optional segment in the definition of DTD. The fraction of people without a homepage is rather high so that this query also presents a challenging path traversal to non-clustering systems.

The speedup curves of Q_{17} under two partition strategies were shown in Fig.11–Fig.15. When the documents are small, PSPIB and $\mathcal{WIN}$ have similar

speedup but $\mathcal{WIN}$ is more stable. With the increasing of document size, nevertheless, it is obviously that the speedup of $\mathcal{WIN}$ is better than PSPIB. The reason is that this query contains twig query. There are some efficient twig join methods [3, 5, 6] in single processor enviroment. This query is decomposed into several subqueries through query rewriting, when the documents are small, the communication cost is low and only a nice distinction in speedup, while with the increasing size of documents, PSPIB has large communication cost and $\mathcal{WIN}$ has little communication cost. Thus $\mathcal{WIN}$ has better speedup performance than PSPIB in this case.

Both $\mathcal{WIN}$ and PSPIB's scaleup curves of Q_{17} were shown in Fig.16. From it, we can clearly see that: with the increasing of the site number, the scaleup curve of PSPIB changes irregularly. As to the scaleup of $\mathcal{WIN}$, the case of three processors is a exception point, $\mathcal{WIN}$ has better tendency except this point. Therefore, $\mathcal{WIN}$ has better scaleup performance in this case.

Besides above query expressions, the query set includes many complex query expressions which include multiple query paths. Finally we give the performance analysis of Q_{20}. This query is about aggregation. The following query computes a simple aggregation by assigning each person to a category. Note that the aggregation is truly semi-structured as it also includes those persons for whom the relevant data is not available. It is shown below:

5 Conclusions and Future Work

In this paper, we proposed the concept of $INode$ and developed a novel workload-aware data placement strategy, $\mathcal{WIN}$, which partitions XML data tree in parallel XML database systems. Extensive experimental results show that our $\mathcal{WIN}$ strategy has much better speedup and scaleup performance than previous data placement strategy PSPIB which is based on XML inherent structure.

As for future work, we plan to further investigate workload-aware data placement strategy and exploit different strategies for different requirements. There are also many challenges in expanding this strategy about replacement problem when the XML document is updated or query set is changed.

Acknowledgement

This work is partially supported by the CUHK strategic grant(numbered 4410001).

References

[1] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *Proceedings of ICDE'02*, 2002.

[2] H. Boral et al. Prototyping Bubba, a highly parallel database system. *IEEE Transactions on Knowledge and Data Engineering*, 2:4–24, 1990.

[3] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In *Proceedings of SIGMOD'02*, 2002.

[4] C.-Y. Chan, M. Garofalakis, and R. Rastogi. RE-Tree: An efficient index structure for regular expressions. In *Proceedings of VLDB'02*, 2002.

[5] Z. Chen, H. V. Jagadish, F. Korn, N. Koudas, S. Muthukrishnan, R. Ng, and D. Srivastava. Counting twig matches in a tree. In *Proceedings of ICDE'01*, 2001.

[6] H. Jiang, W. Wang, H. Lu, and J. X. Yu. Holistic twig joins on indexed XML documents. In *Proceedings of VLDB'03*, 2003.

[7] H. Lu, G. Wang, G. Yu, Y. Bao, J. Lv, and Y. Yu. XBase: Making your gigabyte disk files queriable. In *Proceedings of SIGMOD'02*, 2002.

[8] J. Lv, G. Wang, J. X. Yu, G. Yu, H. Lu, and B. Sun. Performamce evaluation of a DOM-based XML database: Storage, indexing and query optimization. In *Proceedings of WAIM'02*, 2002.

[9] D. Saccà and G. Wiederhold. Database partitioning in a cluster of processors. In *Proceedings of VLDB '83*, 1983.

[10] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *Proceedings of VLDB'02*, 2002.

[11] Teradata Corporation. *DBC/1012 Data Base Computer System Manual*. Teradata Corporation Document, 1985. Release 2.0.

[12] G. Yu, K. Kaneko, G. Bai, and A. Makinouchi. Transaction management for a distributed store system WAKASHI design, implementation and performance. In *Proceedings of ICDE'96*, 1996.

[13] Y. Yu, G. Wang, G. Yu, G. Wu, J. Hu, and N. Tang. Data placement and query processing based on RPE parallelisms. In *Proceedings of COMPSAC'03*, 2003.